

---

# References, Local State, and Objects

Mattox Beckman

beckman@iit.edu

Illinois Institute of Technology

## Objectives

---

## §0 Objectives

The lack of mutable variables gives us the ability to perform many analyses using mathematics. In this lecture we talk about *equational reasoning* and references, and see techniques for limiting the scope of the state to improve the reliability of your code.

- Be able to explain equational reasoning and give an example.
- Know the syntax of references in OCaml.
- Know the tradeoffs between imperative and functional features.
- Know the constructions to define a function with local state.
- Be able to state the benefits of local state and give an example.
- Be able to use tuples to allow multiple functions access to the same state.

## More Objectives

---

## §0 Objectives

In this lecture we also extend the idea of local state to create a simple implementation of objects, and discuss its limitations. We will also show the message dispatch model of objects, which allows for inheritance and virtual functions.

Your objectives:

- Be able to explain what an object is.
  - Know how to implement an object using records and HOFs.
  - Know how to implement an object using a message dispatcher.
  - Be able compare the record and dispatcher models.
  - Major goal 1: be able to simulate objects in a language lacking them.
  - Major goal 2: understand how objects work “under the hood”.
-

**Definition****§1 Equational Reasoning**

---

The rule of *referential transparency*:

$$\frac{e_1 \Downarrow v \quad e_2 \Downarrow v \quad f \ e_1 \Downarrow w}{f \ e_2 \Downarrow w}$$

- If you have two expressions that evaluate to be the same thing then you can use one for the other without changing the meaning of the whole program.
- e.g.  $f(x) + f(x) == 2 * f(x)$
- You can prove this by induction, using the natural semantic rules from the previous lectures.

- You can use equational reasoning to make the following equivalence:

$$f(\text{if } e_1 \text{ then } e_2 \text{ else } e_3) \equiv \text{if } e_1 \text{ then } f(e_2) \text{ else } f(e_3)$$

1

2

`x * (if foo then 20 / x else 23 / x) equivalent to  
if foo then 20 else 23     (well, mostly)`

- You have the basis now of many compiler optimization opportunities!

# A Complication

## §1 Equational Reasoning

---

```
1 # let counter = something
2 val counter : unit -> int = <fun>
3 # counter ();;
4 - : int = 1
5 # counter ();;
6 - : int = 2
7 # counter ();;
8 - : int = 3
9 #
```

- Can we still use equational reasoning to talk about programs now?

## A Counterexample

## §1 Equational Reasoning

---

●  $f(x) + f(x) == 2 * f(x)$

```
1 # 2 * counter ();  
2 - : int = 8  
3 # counter () + counter ();  
4 - : int = 11
```

● Congratulations. You just broke mathematics.

## Transition Semantics

$\text{ref } v \rightarrow \$i$ , where  $\$i$  is a free location in the state, initialized to  $v$ .

$! \$i \rightarrow v$ , if state location  $\$i$  contains  $v$

$\$i := v \rightarrow ()$ , and state location  $\$i$  is assigned  $v$ .

$() ; e \rightarrow e$

Note that references are different than pointers: once created, they cannot be moved, only assigned to and read from.



$\frac{e \Downarrow v}{\text{ref } e \Downarrow \$i}$ , where  $\$i$  is a free location in the state, initialized to  $v$ .

$\frac{e \Downarrow \$i}{!e \Downarrow v}$ , if state location  $\$i$  contains  $v$ .

$\frac{e_1 \Downarrow \$i \quad e_2 \Downarrow v}{e_1 := e_2 \Downarrow ()}$ , and location  $\$i$  is set to  $v$ .

$\frac{e_1 \Downarrow () \quad e_2 \Downarrow v}{e_1; e_2 \Downarrow v}$

```
1 # let ct = ref 0;;
2 val ct : int ref = {contents=0}
3 # let counter () =
4     ct := !ct + 1;
5     !ct;;
6 val counter : unit -> int = <fun>
7 # counter ();;
8 - : int = 1
9 # counter ();;
10 - : int = 2
```

## Bad Things for Counter

---

## §2 References

`ct` is globally defined. Two bad things could occur because of this.

1. What if you already had a global variable `ct` defined?
  - Correct solution: use modules.
2. The Stupid User<sup>TM</sup> might decide to change `ct` just for fun.
  - Now your counter won't work like it's supposed to...
  - Now you can't change the representation without getting tech support calls.
  - Remember the idea of *abstraction*.

## Conclusions about State

---

## §2 References

State is bad because:

- it breaks our ability to use equational reasoning
- users can get to our global variables and change them without permission

State is good because:

- Certain constructs are almost impossible without state (e.g., Graphs)
- Our world is a stateful one

## Review of Scope

---

```
1 let x = 10;;  
2  
3 let foo y = match y with  
4   | 0,b -> let c = b * b in  
5             let d = c * c in  
6             b * c * d  
7   | a,b -> map (fun z -> z + a + x) [a;b]
```

- `x` exists from line 2–7.
- `y` exists from line 3–7.
- `b` exists from line 4–6.
- `c` exists from line 5–6.
- `d` exists on line 6 only.
- `a` and `b` exist on line 7 only.
- `z` exists on line 7, after the `fun z` until the `)`.

```
1 # let counter =  
2   let ct = ref 0 in  
3   fun () -> ct := !ct + 1; !ct;;  
4 val counter : unit -> int = <fun>  
5 # counter ();;  
6 - : int = 1  
7 # counter ();;  
8 - : int = 2
```

- This protects `ct`, making it available only to `counter`.

## Random Number Generators

## §4 Local State

---

```
1 # let mkRandom s =  
2     fun () -> s := (!s * 9 + 5) mod 1024; !s;;  
3 val mkRandom : int ref -> unit -> int = <fun>  
4 # let rnd0 = mkRandom (ref 1);;  
5 val rnd0 : unit -> int = <fun>  
6 # rnd0 ();;  
7 - : int = 14  
8 # rnd0 ();;  
9 - : int = 131  
10 # rnd0 ();;  
11 - : int = 160
```

- In this version we pass the reference into the function rather than creating our own.

```
1 # let (counter, reset) =  
2     let ct = ref 0 in  
3         (fun () -> ct := !ct + 1; !ct),  
4         (fun nv -> ct := nv));;  
5 val counter : unit -> int = <fun>  
6 val reset : int -> unit = <fun>  
7 # counter ();;  
8 - : int = 1  
9 # reset 5;;      (* This trick brought to you by *)  
10 - : unit = ()   (* higher order functions, tuples, *)  
11 # counter ();;  (* and the principle of orthogonality. *)  
12 - : int = 6
```



## Passing Counters Around

---

### §4 Local State

```
1 # let enumerate lst (ctfun, rsfun) =  
2     rsfun 0;  
3     List.map (fun x -> (ctfun (), x)) lst;;  
4 val enumerate : 'a list ->  
5     (unit -> 'b) * (int -> 'c) -> ('b * 'a) list = <f  
6 # enumerate ["hello"; "there"; "class"]  
7     (counter, reset);;  
8 - : (int * string) list = [1, "hello"; 2, "there";  
9                             3, "class"]  
10 #
```

- We can give the counter to another function.
- This is not good. Why not?

**Activity**

---

1. Supposing you wanted a counter that did *not* use references, how would you go about writing it?
2. The random number function generator does not have a way to reset the state. We would also like to be able to ask “what was the last random number generated” without changing the seed. Write a (group of) functions to do this.
3. Suppose we want a more generic way to represent counters—in fact, suppose you want *several* counters in your program, each with an init and a reset. You could just repeat the code several times, but there are serious flaws to that approach. What are they? How might you go about fixing them?

**Answer to 2**

---

```
1 # let mkRandom init =  
2   let seed = ref init in  
3   (( fun () -> seed := !seed * 4; !seed),  
4     ( fun ns -> seed := ns),  
5     ( fun () -> !seed) );;  
6   val mkRandom : int -> (unit -> int) * (int -> unit)  
7 # let (rnd, reset, last) = mkRandom 4;;  
8 val rnd : unit -> int = <fun>  
9 val reset : int -> unit = <fun>  
10 val last : unit -> int = <fun>
```

## Preliminaries

## §6 Motivating Objects

---

- We will use the following functions during our discussion....

```
1 let pi1 (x,y) = x
2 let pi2 (x,y) = y
3 let report (x,y) = print_string "Point: ";
4                   print_int x;
5                   print_string ", ";
6                   print_int y;
7                   print_newline ()
8 let movept (x,y) (dx,dy) = (x+dx,y+dy)
```

**Point****§6 Motivating Objects**

---

Here is an example of a point using local state.

```
1 let mkPoint myloc =  
2   let myloc = ref myloc in  
3     ( myloc,  
4       (fun () -> pi1 !myloc),  
5       (fun () -> pi2 !myloc),  
6       (fun () -> report !myloc),  
7       (fun dl -> myloc := movept !myloc dl) )
```

- This defines a tuple of functions that share a common state.
- It is cumbersome to use.

```
let (lref, getx, gety, show, move) = mkPoint (2,4);;
```

---

**Improvement: Use records.****§6 Motivating Objects**

---

```
1 type point = {  
2   loc : (int * int) ref;  getx : unit -> int;  
3   gety : unit -> int;  draw : unit -> unit;  
4   move : int * int -> unit;  
5 }  
6 let mkrPoint newloc =  
7   let myloc = ref newloc in  
8   { loc = myloc;  
9     getx = (fun () -> pi1 !myloc);  
10    gety = (fun () -> pi2 !myloc);  
11    draw = (fun () -> report !myloc);  
12    move = (fun dl -> myloc := movept !myloc dl) }
```

What is an object?

- Data and functions are grouped together.
- Functions have their own local state.
- Objects can send and receive *messages*.
- Objects can refer to *themselves*.

This has a profound effect on the way programs are written.  
Remember the basic premise of this course: how you think about data has a great impact on the way a program is written.

- How is the `mkrPoint` example like an object?
- How is the `mkrPoint` example not like an object?

## Adding Self

---

By the way, this lecture is really about recursion.

```
1 let mkPoint newloc =  
2   let rec this =  
3     { loc = ref newloc;  
4       getx = (fun () -> pi1 !(this.loc));  
5       gety = (fun () -> pi2 !(this.loc));  
6       draw = (fun () -> report !(this.loc));  
7       move = (fun dl ->  
8         this.loc := movept !(this.loc) dl)}  
9   in this;;
```

We can store “this” explicitly in the record if we want.



## Memory

---

- The record `point` contains references to the fields. If you copy a `point`, the data does **not** get copied!

```
1 # let p1 = mkPoint (4,7);;
2 val p1 : point = {loc={contents=4, 7}; ...}
3 # let p2 = mkPoint (6,2);;
4 val p2 : point = {loc={contents=6, 2}; ...}
5 # let p3 = p1;;
6 val p3 : point = {loc={contents=4, 7}; ...}
7 # p1.move (5,5);;
8 - : unit = ()
9 # p3;;
10 - : point = {loc={contents=9, 12}; ...}
```

**So far...****§8 Dispatching**

---

We used a record to implement a type for points.

Advantages:

- Every method had its own name and type.
- Simple syntax for manipulating the object.
- It's fast: we know at compile time which method is being called.

Disadvantages:

- Inheritance is very difficult with this model.
- Adding a new message type means updating *everything*.

## Message Dispatching

---

## §8 Dispatching

Last time we said that an object is a kind of data that can *receives messages* from the program or other objects.

- Q: How do we normally represent messages?
- A: With strings!

Let a point object be a function which takes a string and returns an appropriate function matching that string.

- Question: Suppose  $p$  is our point object. What will be its type?

**mkPoint****§8 Dispatching**

---

```
1 let mkPoint x y =  
2   let x = ref x in  
3   let y = ref y in  
4   fun st ->  
5     match st with  
6     | "getx" -> (fun _ -> !x)  
7     | "gety" -> (fun _ -> !y)  
8     | "movx" -> (fun nx -> x := !x + nx; nx)  
9     | "movy" -> (fun ny -> y := !y + ny; ny)  
10    | _ -> raise (Failure "Unknown message.")
```

All methods now have to have type `int -> int`.

## Subclassing

## §8 Dispatching

---

- Warmup exercise: How would we add a `report` method?
- Another one: How would we add `this` support?

Let's say we want a `fastpoint`, which moves twice as fast as the original `point`. What does it mean for `fastpoint` to be a *subclass* of `point`?

- `fastpoint` should respond to the same messages.
  - It may override some of them.
  - It may add its own.
  - It may **not** remove any methods.
- The `fastpoint` object will need access to some of the data in `point`.

- Two entities involved: the superclass (`point`) and the subclass (`fastpoint`).
- `fastpoint` needs to create an instance of `point`.
- `point` construction needs to return the “public” data to `fastpoint`.
- `fastpoint` returns a dispatcher:
  - if the `fastpoint` dispatcher can handle a message, it does.
  - Otherwise, it *sends the message* to `point`.

**Code: point****§8 Dispatching**

```
1 let mkSuperPoint x y =  
2   let x = ref x in  
3   let y = ref y in  
4   ((x,y),  (* This part returns the local state *)  
5   fun st ->  
6     match st with  
7     | "getx" -> (fun _ -> !x)  
8     | "gety" -> (fun _ -> !y)  
9     | "movx" -> (fun nx -> x := !x + nx; nx)  
10    | "movy" -> (fun ny -> y := !y + ny; ny)  
11    | _ -> raise (Failure "Unknown message."));;  
12 val mkSuperPoint : int -> int ->  
13   (int ref * int ref) * (string -> int -> int) = <
```

**Code: fastpoint****§8 Dispatching**

---

```
1 let mkFastpoint x y =  
2   let ((x,y),super) = mkSuperPoint x y in  
3   fun st ->  
4     match st with  
5     | "movx" -> (fun nx -> x := !x + 2 * nx; nx)  
6     | "movy" -> (fun ny -> y := !y + 2 * ny; ny)  
7     | _ -> super st;;
```

- This technique is flexible; we can add methods very easily.
- But it's also slow. Imagine if we had a chain of 20 classes....



- Methods and variables are kept in a table: a fixed location.
- “this” is an implicit argument, allowing only one copy of the function to be needed.
- Virtual methods are kept in a *vtable*, which counts as local data.

Local data for `point` or `fastpoint`:

<code>x</code>	<i>value of x</i>
<code>y</code>	<i>value of y</i>
<code>vtable</code>	<i>pointer to vtable</i>

Vtable for `point`:

<code>movx</code>	<i>pointer to point.movx</i>
<code>movy</code>	<i>pointer to point.movy</i>

(`fastpoint` vtable is similar.) `getx`, etc. is static.

## Discussion

---

- Other languages (i.e., smalltalk) use a technique very similar to this one.
- Java uses the “every object is of type `Object`” technique.
- A strong type system makes it somewhat cumbersome to simulate objects. You either have to:
  - define a new type to encompass all objects, or
  - force all methods to have the same type.
- Important concept: *polymorphism* — when functions can operate on multiple types. (This is different than *overloading* — when multiple functions exist with the same name, but different inputs.)

## Polymorphism Example

---

```
1 # let p1,p2,p3,p4 = mkPoint 2 3, mkPoint 3 2,  
2           mkFastpoint 5 3, mkFastpoint 3 9;;  
3 # List.map (fun pt -> pt "report" 0)  
4           [p1; p2; p3; p4];;  
5 Point: 2,3  points  
6 Point: 3,2  
7 Point: 5,3  fastpoints  
8 Point: 3,9
```

The function passed to `map` will use both `point` and `fastpoint` types.

You have seen polymorphism in the course before.

## Conclusions

---

- Objects have a lot of flexibility, and allow us to create useful abstractions.
- They can be implemented using functions.
- These are useful enough in practice, and difficult enough to implement, that most modern languages now include them, including OCaml. (That's where the O comes from.)